CHR **modulaire avec ask et tell**

François Fages[1], **Thierry Martinez**[1], **Cleyton Rodrigues**[2]

[1] Équipe Projet Contraintes, INRIA Paris–Rocquencourt, France
[2] Universidade Federal de Pernambuco, Recife, Brazil
{francois.fages,thierry.martinez}@inria.fr, cleyton.rodrigues@gmail.com

Résumé

Dans ce papier, nous introduisons une version modulaire du langage *Constraint Handling Rules* (CHR), appelé CHRat pour CHR modulaire avec *ask* et *tell*. Toute contrainte définie dans un composant CHRat peut être réutilisée à la fois dans les règles et les gardes d'un autre composant CHRat pour définir de nouveaux solveurs de contraintes. Contrairement aux travaux précédents sur la modularité de CHR, notre approche est complètement générale. Elle ne repose pas sur une condition de dérivabilité automatique de la vérification de l'implication des gardes, mais sur une discipline de programmation qui invite à définir par des règles CHRat la vérification à la fois de la satisfiabilité (*tell*) et de l'implication (*ask*) pour chaque contrainte. Nous définissons les sémantiques opérationnelles et déclaratives de CHRat, décrivons une transformation des composants CHRat en programmes CHR classiques, et prouvons la préservation de la sémantique. Nous donnons ensuite des exemples de modularisation pour des solveurs de contraintes CHR classiques.

1 Introduction

Le langage *Constraint Handling Rules* a été introduit il y a près de vingt ans comme langage déclaratif pour définir des solveurs de contraintes par des règles de réécriture de multi-ensembles avec gardes, étant donné un système de contraintes noyau prédéfini [10]. Le paradigme de programmation de CHR permet l'implantation de systèmes de contraintes en déclarant des règles de réécriture gardées qui transforment un *store* de contraintes en une forme résolue afin d'en déterminer la satisfiabilité. La forme résolue, atteinte lorsqu'il n'y a plus de transformation applicable, est insatisfiable si elle contient la contrainte $\langle \text{false} \rangle$, et est opérationnellement satisfiable sinon. Une propriété importante, mais non obligatoire, de ces transformations est la *confluence* qui signifie que la forme résolue est toujours indépendante de l'ordre d'application des règles, et constitue ainsi une *forme normale* pour le store de contraintes initial [1].

Depuis lors, CHR a évolué vers un langage de programmation généraliste à base de règles [10] avec des extensions telles que la gestion de la disjonction [3] ou l'introduction de types [5]. Cependant, un des inconvénients majeurs de CHR est l'absence de *modularité*. Une fois qu'un système de contraintes est défini en CHR à partir des contraintes du noyau, ce système de contraintes ne peut pas être réutilisé dans un autre programme CHR en bénéficiant des contraintes ainsi définies comme autant de nouvelles contraintes noyau. La raison de cette difficulté est qu'un programme CHR définit un test pour la satisfiabilité mais il n'en définit pas pour l'implication, pourtant nécessaire pour vérifier les gardes.

Les approches précédentes vis-à-vis de ce problème ont étudié des conditions sous lesquelles il est possible de dériver automatiquement un test d'implication à partir d'un test de satisfiabilité, notamment des conditions reposant sur l'équivalence logique[17] :

$$\mathcal{X} \models c \rightarrow d \text{ si et seulement si } \mathcal{X} \models (c \wedge d) \leftrightarrow c$$

Dans ce papier, nous proposons une version *modulaire* de CHR appelée CHR avec *ask* et *tell*, et notée CHRat [1]. Ce paradigme s'inspire du paradigme de programmation concurrente par contraintes [16, 15]. La discipline de programmation proposée dans CHRat pour la programmation modulaire de solveurs de contraintes demande, pour chaque contrainte introduite par le programmeur, de définir de règles de simplification et de propagation qui réécrivent des jetons de contrôle (*control token*) *ask* en des jetons de

[1] L'implantation de CHRat et des exemples de définition de hiérarchies de solveurs de contraintes sont disponibles à `http://contraintes.inria.fr/~tmartine/chrat`

contrôle *entailed*. La nécessité de définir des solveurs pour les *asks* et *tells* est déjà établie pour les implantations de systèmes de contraintes noyau [8] ; la discipline que nous proposons consiste à étendre ce principe aux solveurs CHR eux-mêmes. Une contrainte de garde $c(\vec{v})$ dans une instance de règle `R` est *opérationnellement impliquée* dans un store de contraintes contenant le jeton de contrôle `ask(K,` $c(\vec{v})$`)` lorsque sa forme résolue contient le jeton `entailed(K,`$c(\vec{v})$`)`, où `K` est une variable fraîche utilisée pour associer les jetons de contrôle à la règle `R`. Ceci nous permet de programmer des vérifications d'implications arbitrairement complexes par des règles, alors que ces vérifications reposent jusqu'à présent sur des programmes impératifs événementiels [7]. Avec cette discipline de programmation, les contraintes CHRat peuvent être réutilisées à la fois dans les règles et les gardes d'autres composants pour définir de nouveaux solveurs.

Dans la prochaine section, nous commençons par illustrer cette approche par un exemple simple. La section 3 énonce un certain nombre de définitions formelles sur CHR présenté comme un langage de réécriture de multi-ensembles, avec sa sémantique déclarative en logique linéaire (et en logique classique dans le cas de la réécriture ensembliste). Ensuite la section 4 introduit les sémantiques opérationnelles et déclaratives pour CHRat. La section 5 décrit la transformation de programmes CHRat en programmes CHR classiques et prouve sa correction. La section 6 illustre ensuite par des exemples la définition hiérarchique de solveurs de contraintes complexes. Enfin, nous concluons par une discussion sur la simplicité et l'expressivité de cette approche et des quelques limitations que nous rencontrons pour le moment.

2 Exemple introductif

2.1 Composants CHRat pour `leq/2` et `min/3`

Nous considérons pour commencer le programme CHR classique définissant une contrainte qui décrit une relation d'ordre.

Exemple 1 *Le solveur de satisfiabilité est défini par les quatre règles ci-dessous. Les trois premières règles traduisent les axiomes des relations d'ordre, et la dernière élimine les contraintes* `leq` *en double.*

```
leq(X,X) <=> true.
leq(X,Y), leq(Y,X) <=> X = Y.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
leq(X,Y) \ leq(X,Y) <=> true.
```

En CHRat, un solveur de contraintes doit définir des règles pour vérifier l'implication. Pour `leq(X,Y)`, ces règles réécrivent le jeton `ask(K,leq(X,Y))` en `entailed(K)`. `K` est une variable qui associe les jetons de contrôle aux instances des règles qui les ont générés, de façon à empêcher d'autres règles partageant la même garde de capturer ces jetons. Dans cet exemple, puisque le store est transitivement clos, l'implication de `leq(X,Y)` est directement observable dans le store par une simple règle si $X \neq Y$. Le cas réflexif est géré par une règle supplémentaire.

```
leq(X,Y)\ask(K,leq(X,Y)) <=>
    entailed(K,leq(X,Y)).
ask(K,leq(X,X)) <=>
    entailed(K,leq(X,X)).
```

Le solveur de satisfiabilité et le solveur d'implication pris ensembles définissent un composant CHRat pour la contrainte `leq(X,Y)`. Pour séparer les espaces de noms des modules CHRat, l'implantation repose sur un simple mécanisme de séparation de noms par différentiation des atomes (*atom-based*) [14] : les contraintes CHR exportées sont préfixées par le nom du composant et les contraintes CHR internes sont préfixées de façon à éviter les collisions [14]. De tels composant peuvent être réutilisés pour définir de nouveaux solveurs en utilisant la contrainte `leq(X,Y)` à la fois dans les règles et les gardes.

Exemple 2 *Le composant ci-dessous définit un solveur pour la contrainte minimum* `min(X,Y,Z)`*, établissant que* `Z` *est la valeur minimale parmi* `X` *et* `Y` *:*

```
component min_solver.
import leq/2 from leq_solver.
export min/3.
min(X,Y,Z) <=> leq(X,Y) | Z=X.
min(X,Y,Z) <=> leq(Y,X) | Z=Y.
min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).

min(X,Y,Z) \ ask(K, min(X,Y,Z)) <=>
    entailed(K, min(X,Y,Z)).
ask(K, min(X, Y, X)) <=> leq(X, Y) |
    entailed(K, min(X,Y,Z)).
ask(K, min(X, Y, Y)) <=> leq(Y, X) |
    entailed(K, min(X,Y,Z)).
```

2.2 Transformation vers un programme CHR classique

Dans CHR, les gardes sont restreintes aux contraintes du noyau [10]. Les composants CHRat sont transformés en programmes CHR classiques par une transformation de programme qui :

- retire des gardes toutes les contraintes définies par l'utilisateur ;
- renomme les prédicats `ask(K,c(...))` en `ask_c(K,...)` avec un argument supplémentaire pour la variable d'association `K` ;

– introduit pour chaque règle CHRat une contrainte CHR id_n qui relie la variable d'association aux instances des variables de la tête ;
– sépare les espaces de noms.

Par exemple, le solveur CHRat `min` est transformé en le programme CHR suivant (modulo séparation de l'espace de noms par soucis de concision) :

```
min(X,Y,Z)\ask_min(K,X,Y,Z)==>
    entailed_min(K,X,Y,Z).
min(X,Y,Z)==>ask_leq(K,X,Y),id1(K,X,Y,Z).
id1(X,Y,Z,K),min(X,Y,Z),
    entailed_leq(K,X,Y)<=>Z=X.
min(X,Y,Z)==>ask_leq(K,Y,X),id2(K,X,Y,Z).
id2(X,Y,Z,K),min(X,Y,Z),
    entailed_leq(K,Y,X)<=>Z=Y.
min(X,Y,Z)==>leq(Z,X),leq(Z,Y).
ask_min(X,Y,X,K)==>
    ask_leq(K0,X,Y),id3(K0,X,Y,K).
id3(K0,X,Y,K),ask_min(K,X,Y,X),
    entailed_leq(K0,X,Y)<=>
    entailed_min(K,X,Y,X).
ask_min(X,Y,Y,K)==>
    ask_leq(K0,Y,X),id4(K0,X,Y,K).
id4(K0,X,Y,K),ask_min(X,Y,Y,K),
    entailed_leq(K0,Y,X)<=>
    entailed_min(K,X,Y,Y).
```

2.3 Variables quantifiées existentiellement dans les gardes

En CHRat, comme en CHR, les variables qui apparaissent dans une garde sans apparaître dans la tête de la règle nécessitent un traitement spécifique. Par exemple, considérons la règle CHRat suivante pour éliminer les éléments non minimaux :

```
number(A), number(B) <=> min(A,B,C) |
    number(C).
```

Le solveur d'implication défini précédemment ne peut pas détecter l'implication d'une telle garde.

En CHRat, les variables `V` quantifiées existentiellement dans les gardes sont marquées par des jetons de contrôle `exists(K,V)`, avec `K` la variable d'association introduite par l'instance de la règle dont la garde est à vérifier. Les règles pour `min/2` dans l'exemple 2 sont ainsi complétées avec des règles supplémentaires pour instancier les variables quantifiées existentiellement de la manière suivante :

```
ask(K,min(X,Y,Z)),exists(K,Z) <=>
    leq(X,Y) |
    Z=X, entailed(K, min(X,Y,Z)).
ask(K,min(X,Y,Z)),exists(K,Z) <=>
    leq(Y,X) |
        Z=Y, entailed(K, min(X,Y,Z)).
min(X,Y,M) \ ask(K, min(X,Y,Z)),
    exists(K,Z) <=> leq(X,Y) |
        Z=M, entailed(K, min(X,Y,Z)).
```

Toutes ces règles ajoutent, en plus du jeton `entailed`, la contrainte elle-même, bien qu'elle soit déjà impliquée, dans le store, ceci afin de contraindre les variables existentiellement quantifiées de satisfaire la contrainte. Ceci généralise la solution proposée dans [1] pour la sémantique opérationnelle de CHR pour les contraintes noyau aux contraintes définies par l'utilisateur.

3 Préliminaires sur CHR

Notations

Pour toute fonction f et tout sous-ensemble $X \subseteq \mathrm{dom}(f)$, nous notons $f(X) = \{f(x) \mid x \in X\}$ et cette notation s'étend aux fonctions n-aires par produit cartésien. Pour tout ensemble E, un *multi-ensemble* M sur E est représenté par une fonction de multiplicité $C_M : E \to \mathbb{N}$. Le *support* de M est $\sup(M) = \{e \in E \mid C_M(e) \geq 1\}$. Un multi-ensemble est fini si son support est fini. Un multi-ensemble est (identifié à) un ensemble lorsque $\forall e \in E, C_M(e) \leq 1$. Soit M et M' deux multi-ensembles sur E, la *somme multi-ensembliste* $M \oplus M'$ est définie par $C_{M\oplus M'} : e \mapsto C_M(e) + C_{M'}(e)$ et la *différence multi-ensembliste* $M \ominus M'$ est définie par $C_{M\ominus M'} : e \mapsto \min(C_M(e) - C_{M'}(e), 0)$. $M \subseteq M'$ lorsque pour tout $e \in E$, $C_M(e) \leqslant C_{M'(e)}$. Une *fonction multi-ensembliste* f de M dans M' ([13]) est représentée par une fonction $\sigma_f : E \to E$ telle que $\sigma_f(\sup(M)) \subseteq \sup(M')$: pour tout $N \subseteq M$, $f(N)$ désigne le multi-ensemble défini par $C_{f(N)} = C_N \circ \sigma_f^{-1}$. f est une *injection* si σ_f est une permutation et $f(M) \subseteq M'$. Une fonction multi-ensembliste f' de $N \supseteq M$ dans M' est une *extension* de f si $\sigma_{f'}$ et σ_f coïncident sur $\sup(M)$.

Syntaxe

Les programmes CHR sont construits autour de trois signatures disjointes : une signature Σ pour les symboles de constantes et de fonctions, une signature Π pour les symboles de prédicats définis par l'utilisateur et une signature $\Pi_{\mathcal{X}}$ pour les symboles de prédicats des contraintes du noyau, $\Pi_{\mathcal{X}}$ est supposée contenir le prédicat d'égalité $=$ et les constantes true et false. Les contraintes définies par l'utilisateur sont les propositions atomiques $\mathcal{A}$ sur Π.

Le langage des contraintes noyau est un fragment $\mathcal{L}$ de Σ-$\Pi_{\mathcal{X}}$-formules logiques du premier ordre, clos par conjonction et quantification existentielle. Les contraintes noyau sont interprétées sur une structure fixée $\mathcal{X}$. Une contrainte $c \in \mathcal{L}$ est *$\mathcal{X}$-satisfiable* si $\mathcal{X} \models \exists \vec{x}(c)$ où $\vec{x}$ est l'ensemble des variables libres de c,

dénoté $V(c)$. Une contrainte c *implique* une contrainte d de $\mathcal{X}$ si $\mathcal{X} \models \forall \vec{x}(c \rightarrow d)$ où $\mathcal{X} \models \forall \vec{x}(c \rightarrow d)$ avec $\vec{x} = V(c) \cup V(d)$. Nous supposons que la satisfiabilité et l'implication sont décidables dans $\mathcal{X}$.

Un programme CHR($\mathcal{X}$) P est une suite finie de règles de réécriture de la forme suivante : $H \backslash H' \Leftrightarrow G \mid B$ où les *têtes* H et H' sont des multi-ensembles finis de contraintes utilisateur, la *garde* G est une conjonction de contraintes noyau, et le *corps* B est la paire $(B_{\mathcal{X}}, B_u)$ où $B_{\mathcal{X}}$ est une conjonction de contraintes noyau et B_u est un multi-ensemble de contraintes utilisateur. Une *règle de simplification* est une règle où H' est vide, et est notée $H' \Leftrightarrow G \mid B$. Une *règle de propagation* est une règle où H est vide, et est notée $H \Rightarrow G \mid B$. Une *règle de simpagation* est une règle où H et H' sont tous deux non-vides. H et H' ne peuvent pas être simultanément vides.

Sémantique opérationnelle

Soit $\rightarrow$ la relation de transition sur les *états* (T, c) avec un store contenant le multi-ensemble des contraintes définies par l'utilisateur T et une contrainte noyau c. Une règle s'applique à un état (T, c) si sa tête correspond à un multi-ensemble de T et si sa garde est impliquée par c. Formellement :

$$(T, c) \rightarrow (\langle T \ominus \pi(H') \oplus B_u \rangle, \langle c \wedge c_m \wedge B_{\mathcal{X}} \rangle)$$

s'il existe une règle $H \backslash H' \Leftrightarrow G \mid B_{\mathcal{X}}, B_u$ dans P (avec les variables renommées en dehors de (T, c)) et une injection π de $T' = H \oplus H'$ dans T telle que la contrainte $c_m = \langle (\bigwedge_{t \in \sup(T')} t = \sigma_\pi(t)) \wedge G \rangle$, dite de correspondance (*matching*) soit impliquée, c'est-à-dire $\mathcal{X} \models \forall \vec{x}(B_{\mathcal{X}} \rightarrow \exists \vec{y}(c_m))$ où $\vec{x} = V(B_{\mathcal{X}})$ et $\vec{y} = V(c_m) \setminus V(c)$.

En considérant les dérivations à partir d'un état initial (T_0, c_0), aussi appelée *requête*, l'observation de tous les états accessibles nous conduit à considérer la sémantique opérationnelle suivante :

$$\mathcal{O}^a_P(T_0, c_0) = \{(T, c) \mid (T_0, c_0) \xrightarrow{*} (T, c)\}$$

avec $\xrightarrow{*}$ désignant la clôture réflexive et transitive de $\rightarrow$, et $\not\rightarrow$ le fait qu'aucune transition ne s'applique.

Sémantiques déclaratives

CHR possède deux sémantiques déclaratives : l'une en logique classique [10] pour les programmes linéaires à gauche et l'une en logique linéaire sans restriction [4]. Une règle est dite *linéaire à gauche* si toute contrainte utilisateur ne correspond qu'à au plus une contrainte dans la tête de cette règle. Un programme est linéaire à gauche dès lors que toutes les règles le sont.

En logique classique, un multi-ensemble M de contraintes est interprété par la conjonction $M^\dagger = \langle \bigwedge_{c \in \sup(M)} c \rangle$ de ses éléments. Pour chaque règle CHR :

$$\begin{aligned}&\left(H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u\right)^\dagger = \\ &\quad \forall \vec{x}(\exists \vec{y}(G) \wedge H^\dagger \rightarrow (H'^\dagger \leftrightarrow \exists \vec{z}(G \wedge B_{\mathcal{X}} \wedge B_u^\dagger)))\end{aligned}$$

où $\vec{x} = V(H_0, H_1)$, $\vec{y} = V(G) \setminus V(H_0, H_1)$ et $\vec{z} = V(G, B_{\mathcal{X}}, B_u) \setminus V(H_0, H_1)$. L'interprétation logique d'un programme $(P)^\dagger$ est la conjonction logique des interprétations de chacune des règles de P. La sémantique logique de la requête q est :

$$\mathcal{L}_P(q) = \{c \mid (P)^\dagger \models_{\mathcal{X}} q \rightarrow c\}$$

$\mathcal{L}_P(q)$ est clos par implication logique dans cette traduction. Pour tout ensemble S de fait logique, nous notons $\downarrow S = \{c' \mid \exists c \in S, \models_{\mathcal{X}} c \rightarrow c'\}$. Dès lors, $\mathcal{L}_P(q) = \downarrow \mathcal{L}_P(q)$.

Soit V un ensemble de variables libres sur la requête (T_0, c_0). Les états sont interprétés comme $(T, c)^\dagger = \langle \exists \vec{x}((T)^\dagger \wedge c) \rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. La notation est étendue aux ensembles S d'états : $S^\dagger = \{(T, c)^\dagger \mid (T, c) \in S\}$.

Théorème 1 *Pour tout programme* CHR($\mathcal{X}$) *linéaire à gauche P et toute requête (T_0, c_0) :*

$$\downarrow (\mathcal{O}^a_P(T_0, c_0))^\dagger \subseteq \mathcal{L}_P((T_0, c_0)^\dagger)$$

Cette formulation donne seulement un résultat de correction : $\mathcal{L}_P((T_0, c_0)^\dagger) \subseteq \downarrow (\mathcal{O}^a_P(T_0, c_0))^\dagger$ n'est pas vrai en général. Par exemple, considérons le programme $\{\texttt{a <=> b. a <=> c.}\}$: $\mathcal{L}_P((\texttt{a}, \top)^\dagger) = \downarrow\{\texttt{a} \wedge \texttt{b} \wedge \texttt{c}\}$ alors que $\downarrow (\mathcal{O}^a_P(\texttt{a}, \top))^\dagger = \{\texttt{a}, \texttt{b}, \texttt{c}, \top\}$. Un résultat de complétude plus faible est vérifié : $\mathcal{L}_P((T_0, c_0)^\dagger) \subseteq \mathcal{L}_P((\mathcal{O}^a_P(T_0, c_0))^\dagger)$: dans ce cas, la propriété est une conséquence directe de $(T_0, c_0)^\dagger \in \mathcal{L}_P((\mathcal{O}^a_P(T_0, c_0))^\dagger)$.

La sémantique classique n'est pas correcte si P n'est pas linéaire à gauche. Par exemple, avec la simple règle `a, a <=> b`, nous avons $\mathcal{L}_P((\texttt{a}, \top)^\dagger) = \downarrow\{\texttt{a} \wedge \texttt{b}\}$ tandis que $\downarrow (\mathcal{O}^a_P(\texttt{a}, \top))^\dagger = \downarrow\{\texttt{a}\} \neq \downarrow\{\texttt{a} \wedge \texttt{b}\}$. De plus, cette sémantique logique identifie trop de programmes (par exemple, $\{\texttt{a <=> b.a <=> c.}\}$ et $\{\texttt{a <=> b, c.}\}$).

Pour pallier à ces limitations, une sémantique déclarative exprimée dans la logique linéaire de Girard [12] a été développée [4] pour tout programme CHR. Les contraintes noyau sur $\mathcal{X}$ sont traduits par la traduction de Girard en logique linéaire en faisant usage de l'opérateur *bien sûr* noté ! [12]. Un multi-ensemble M de contraintes est interprété par la conjonction multiplicative des contraintes $M^{\dagger\dagger} = \langle \bigotimes_{c \in M} c \rangle$. Pour toute

règle CHR :

$$\big(H\backslash H' \Rightarrow G \mid B\big)^{\dagger\dagger} =$$
$$\langle !\forall\vec{x}(\exists\vec{y}(G^{\dagger\dagger})\otimes H^{\dagger\dagger}\otimes H'^{\dagger\dagger} \multimap \exists\vec{z}(H^{\dagger\dagger}\otimes G^{\dagger\dagger}\otimes B_{\mathcal{X}}^{\dagger\dagger}\otimes B_u^{\dagger\dagger}))\rangle$$

où $\vec{x} = V(H, H')$, $\vec{y} = V(G) \setminus V(H, H')$ et $\vec{z} = V(G, B_{\mathcal{X}}, B_u) \setminus V(H, H')$.

L'interprétation en logique linéaire d'un programme $(P)^{\dagger}$ est la conjonction multiplicative des interprétations des règles de P. La sémantique en logique linéaire d'une requête q est :

$$\mathcal{LL}_P(q) = \{c \mid (P)^{\dagger\dagger} \models_{LL,\mathcal{X}} q^{\dagger\dagger} \multimap c \otimes \top\}$$

$\mathcal{LL}_P(q)$ est clos par implication linéaire. Pour tout ensemble S de faits logiques, nous notons $\downarrow S = \{c' \mid \exists c \in S, \models_{LL,\mathcal{X}} c \multimap c' \otimes \top\}$. Dès lors, $\mathcal{LL}_P(q) = \downarrow\mathcal{LL}_P(q)$.

Soit V l'ensemble des variables libres de la requête (T_0, c_0). Les états sont interprétés comme $(T, c)^{\dagger\dagger} = \langle\exists\vec{x}(T^{\dagger\dagger} \otimes c^{\dagger\dagger})\rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. La notation est étendue aux ensembles S d'états : $S^{\dagger\dagger} = \{(T, c)^{\dagger\dagger} \mid (T, c) \in S\}$.

Théorème 2 ([4, 9]) *Pour tout programme* CHR($\mathcal{X}$) P *et requête* (T_0, c_0) *:*

$$\downarrow(\mathcal{O}_P^a(T_0, c_0))^{\dagger\dagger} = \mathcal{LL}_P((T_0, c_0)^{\dagger\dagger})$$

4 Syntaxe et sémantique des composants CHRat($\mathcal{X}$)

Syntaxe

En CHRat, les jetons de contrôle sont construits sur des symboles frais issus de la signature $\Pi_t = \{\text{ask}/2, \text{exists}/2, \text{entailed}/2\}$, disjointe de Π et $\Pi_{\mathcal{X}}$.

Définition 1 *Un* programme CHRat($\mathcal{X}$) P *est une suite finie de règles* $H\backslash H' \Leftrightarrow G \mid B_{\mathcal{X}}, B_u$ *où* H *est un multi-ensemble d'éléments de* $\mathcal{A}$ *;* H' *est un multi-ensemble d'éléments de* $\mathcal{A} \cup \text{ask}(V, \mathcal{A}) \cup \text{exists}(V, V)$ *;* G *est une formule de* $\mathcal{L}'$ *où* $\mathcal{L}'$ *est l'extension de* $\mathcal{L}$ *contenant les propositions atomiques* $\mathcal{A}$ *et close par conjonction et quantification existentielle ;* $B_{\mathcal{X}}$ *est une formule de* $\mathcal{L}$ *;* B_u *est un multi-ensemble d'éléments de* $\mathcal{A} \cup \text{entailed}(V, \mathcal{A})$*.* H *et* H' *ne peuvent pas être simultanément vides.*

En tant que formule de $\mathcal{L}'$, une garde G est de la forme $G_u \wedge G_{\mathcal{X}}$ où G_u est une conjonction (ou multi-ensemble) de jetons et $G_{\mathcal{X}}$ est une formule de $\mathcal{L}$. L'ensemble des variables qui n'apparaissent que dans G_u est noté $V_G = V(G_u) \setminus V(H, H', G_{\mathcal{X}})$.

Sémantique opérationnelle

Les états sont des triplets (T, c, I), où :
- le store noyau c est une contrainte de $\mathcal{C}$;
- le store utilisateur T est un multi-ensemble sur $\mathcal{A} \cup \text{ask}(V, \mathcal{A}) \cup \text{exists}(V, V) \cup \text{entailed}(V, \mathcal{A})$;
- la table des instances en attente I est une application à support finie des variables vers les instances de règles $(\mathbf{r}, \sigma_\pi)$ où $\mathbf{r}$ est une règle (renommée) et σ_π une permutation de $\mathcal{A}$.

Pour toute table d'instances en attente I, $\sup(I)$ désigne le support de I. Si $K \in V$ est tel que $K \notin \sup(I)$, alors $I \uplus (\mathbf{r}, \sigma_\pi)_K$ désigne l'application I' telle que $\sup(I') = \sup(I) \cup \{K\}$ avec $I'|_{\sup(I)} = I$ et $I'(K) = (\mathbf{r}, \sigma_\pi)$.

Nous introduisons deux types de transitions déclenchables depuis un état (T, c, I) : les suspensions $\xrightarrow{\text{s}}$ et les réveils $\xrightarrow{\text{w}}$.

– la règle de suspension :

$$(T, c, I) \xrightarrow{\text{s}} (\langle T \oplus \text{ask}(K, G_u) \oplus \text{exists}(K, V_G)\rangle, \langle c \wedge \exists(c_m)\rangle, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K\rangle)$$

où $\mathbf{r} = \langle H\backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u\rangle$ est une règle de P renommée avec des variables fraîches par rapport à (T, c, I) et π une injection de $T' = H \oplus H'$ dans T telle que la contrainte de correspondance $c_m = \langle(\bigwedge_{t\in\sup(T')} t = \sigma_\pi(t)) \wedge G_{\mathcal{X}}\rangle$ soit impliquée, c'est-à-dire $\mathcal{X} \models \forall\vec{x}(c \rightarrow \exists\vec{y}(c_m))$ où $\vec{x} = V(c)$ et $\vec{y} = V(c_m) \setminus V(c)$. Alors la transition ajoute la nouvelle instance en attente $(\mathbf{r}, \sigma_\pi)$ à I, indexée par une variable fraîche K.

– la règle de réveil :

$$(T, c, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K\rangle) \xrightarrow{\text{w}} (\langle T \ominus \pi'(H') \ominus \pi'(\text{entailed}(K, G_u)) \oplus B_u\rangle, \langle c \wedge c_m \wedge B_{\mathcal{X}}\rangle, I)$$

où $(\mathbf{r}, \sigma_\pi)$ est une instance en attente de la règle $\mathbf{r} = \langle H\backslash H' \Rightarrow G \mid B_{\mathcal{X}} \wedge B_u\rangle$ et σ_π représente une injection π de $H \oplus H'$ dans T, telle qu'il existe une injection π' de $T' = H \oplus H' \oplus \text{entailed}(K, G_u)$ dans T, qui étende π de telle sorte que la contrainte de correspondance : $c_m = \langle(\bigwedge_{t\in\sup(T')} t = \sigma_{\pi'}(t)) \wedge G_{\mathcal{X}}\rangle$ soit impliquée, c'est-à-dire $\mathcal{X} \models \forall\vec{x}(c \rightarrow \exists\vec{y}(c_m))$ où $\vec{x} = V(c)$ et $\vec{y} = V(c_m) \setminus V(c)$.

Nous définissons $\rightarrow = \xrightarrow{\text{s}} \cup \xrightarrow{\text{w}}$.

Définition 2 *La* sémantique opérationnelle *pour l'observation des états accessibles depuis une requête* $q = T_0 \wedge c_0 \in \mathcal{L}'$ *est, avec* $\vec{x} = V(T_1, c) \setminus V(q)$ *:*

$$\mathcal{O}_P^a[q] = \{\exists\vec{x}(T_1 \wedge c) \mid (T_0, c_0, \emptyset) \xrightarrow{*} (T, c, \_), T_1 = T \cap \mathcal{A}\}$$

La sémantique des stores accessibles $\mathcal{O}^a_P[q]$ est reliée aux sémantiques déclaratives de la prochaine section. Les stores observés sont restreints à $\mathcal{A}$. Soit min le solveur défini dans l'exemple 2, alors $\langle \text{leq}(X,Y) \wedge Z = X \rangle$ est un store accessible et le jeton $\text{ask}(\text{leq}(Y,X))$ introduit dans l'infructueux test d'implication (pour la seconde règle du composant leq) n'est pas exposé.

Sémantique déclarative

Nous définissons la transformation $(\cdot)^*$ des jetons de contrôle et des instances suspendues aux propositions atomiques :

$$(\text{ask}(K, p(\overrightarrow{X})))^* = \text{ask}_p(K, \overrightarrow{X})$$
$$(\text{entailed}(K, p(\overrightarrow{X})))^* = \text{entailed}_p(K, \overrightarrow{X})$$
$$(\text{exists}(K, V))^* = \text{exists}(K, V)$$
$$(((H \backslash H' \Rightarrow G \mid B), \sigma_\pi)_K)^* = \text{id}_{\mathbf{r}}(K, \overrightarrow{V}(\sigma_\pi(H \oplus H')))$$

En logique classique, un multi-ensemble M de contraintes est interprété par la conjonction $M^\ddagger = \langle \bigwedge_{c \in \text{sup}(M)} c^* \rangle$ de ses éléments. Une règle est logiquement interprétée comme :

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger =$$
$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger_{\text{s}} \wedge \big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger_{\text{w}}$$

où les sous-formules *suspend* et *wake* traduisent leur contrepartie opérationnelle :

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger_{\text{s}} = \Big\langle \forall \vec{x} \Big(\exists \vec{y}(G) \wedge H^\ddagger \wedge H'^\ddagger \rightarrow \exists K \big(\text{id}(K, \vec{z}) \wedge \text{exists}(K, V_G) \wedge \big(\bigwedge_{p(\vec{u}) \in \text{sup}(G_u)} \text{ask}_p(K, \vec{u}) \big) \big) \Big) \Big\rangle$$

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, \wedge B_u\big)^\ddagger_{\text{w}} = \Big\langle \forall \vec{x} \forall K \big(\exists \vec{y}(G) \wedge H^\ddagger \rightarrow (H'^\ddagger \wedge \text{id}(K, \vec{z}) \wedge \big(\bigwedge_{p(\vec{u}) \in G_u} \text{entailed}_p(K, \vec{u}) \big) \leftrightarrow \exists \vec{z}(G \wedge B_\mathcal{X} \wedge B^\ddagger_u) \big) \big) \Big\rangle$$

avec $\vec{x} = V(H, H')$, $\vec{y} = V(G) \setminus V(H, H')$ et $\vec{z} = V(G, B_\mathcal{X}, B_u) \setminus V(H, H')$. L'interprétation logique d'un programme $(P)^\ddagger$ est la conjonction logique des interprétations des règles de P.

Définition 3 *La* sémantique en logique classique *d'une règle $q \in \mathcal{L}'$ est :*

$$\mathcal{L}_P[q] = \{c \in \mathcal{L}' \mid (P)^\ddagger \models_\mathcal{X} q \rightarrow c\}$$

Soit V l'ensemble des variables libres de la requête $q = T_0 \wedge c_0 \in \mathcal{L}'$. Les états sont interprétés comme $(T, c, I)^\ddagger = \langle \exists \vec{x}(T^\ddagger \wedge c \wedge (\bigwedge_{K \in \text{sup}(I)} I(K)^*)) \rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. La définition est étendue pour un ensemble d'états S : $S^\ddagger = \{(T, c, I)^{\dagger\dagger} \mid (T, c, I) \in S\}$.

Avec la même restriction que pour CHR, la sémantique opérationnelle de CHRat est correcte vis-à-vis de la sémantique en logique classique. Ce résultat découle du lemme suivant :

Lemme 1 *Si $c_0 \rightarrow c_1$, alors $(P)^\ddagger \models_\mathcal{X} (c_0)^\ddagger \rightarrow (c_1)^\ddagger$.*

Théorème 3 *Pour tout programme* CHRat$(\mathcal{X})$ *linéaire à gauche P et toute requête $q \in \mathcal{L}'$:*

$$\downarrow \mathcal{O}^a_P[q] \subseteq \mathcal{L}_P[q]$$

La sémantique en logique linéaire utilise des propositions atomiques similaires pour coder les jetons de contrôle et les instances suspendues. Un multi-ensemble M de contraintes est interprété par la conjonction multiplicative des contraintes $M^{\ddagger\ddagger} = \langle \bigotimes_{c \in M} c^* \rangle$. Une règle est interprétée en logique linéaire de la manière suivante :

$$\big(H \backslash H' \Rightarrow G \mid B\big)^{\ddagger\ddagger} = \big(H \backslash H' \Rightarrow G \mid B\big)^{\ddagger\ddagger}_{\text{s}} \otimes \big(H \backslash H' \Rightarrow G \mid B\big)^{\ddagger\ddagger}_{\text{w}}$$

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^{\ddagger\ddagger}_{\text{s}} = \Big\langle !\forall \vec{x} \Big(\exists \vec{y}(G^{\ddagger\ddagger}) \otimes H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \multimap \exists K \big(H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \otimes \text{id}(K, \vec{z}) \otimes \text{exists}(K, V_G) \otimes \big(\bigotimes_{p(\vec{u}) \in G_u} \text{ask}_p(K, \vec{u}) \big) \big) \Big) \Big\rangle$$

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^{\ddagger\ddagger}_{\text{w}} = \Big\langle !\forall \vec{x} \forall K \Big(\exists \vec{y}(G^{\ddagger\ddagger}) \otimes H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \otimes \text{id}(K, \vec{z}) \otimes \big(\bigotimes_{p(\vec{u}) \in G_u} \text{entailed}_p(K, \vec{u}) \big) \multimap \exists \vec{z}(H^{\ddagger\ddagger} G^{\dagger\dagger} \otimes B_\mathcal{X} \otimes B^{\ddagger\ddagger}_u) \big) \Big) \Big\rangle$$

avec $\vec{x} = V(H, H')$, $\vec{y} = V(G) \setminus V(H, H')$ et $\vec{z} = V(G, B_\mathcal{X}, B_u) \setminus V(H, H')$. : L'interprétation d'un programme $(P)^{\ddagger\ddagger}$ est la conjonction multiplicative des interprétations des règles de P.

Définition 4 *La* sémantique en logique linéaire *d'une requête $q \in \mathcal{L}'$ est*

$$\mathcal{LL}_P[q] = \{c \in \mathcal{L}' \mid (P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} q \multimap c \otimes \top\}$$

Soit V l'ensemble des variables libres de la requête $q = T_0 \wedge c_0 \in \mathcal{L}'$. Les états sont interprétés comme $(T, c, I)^{\ddagger\ddagger} = \langle \exists \vec{x} (T^{\ddagger\ddagger} \otimes c \otimes (\bigotimes_{K \in I} I(K)^*)) \rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. Cette définition est étendue pour un ensemble d'états S : $S^{\ddagger\ddagger} = \{(T, c, I)^{\ddagger\ddagger} \mid (T, c, I) \in S\}$.

La sémantique opérationnelle de CHRat est correcte et complète vis-à-vis de la sémantique en logique linéaire. Ce résultat découle du lemme suivant :

Lemme 2 *Si* $c_0 \to c_1$, *alors* $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} (c_0)^{\ddagger\ddagger} \multimap (c_1)^{\ddagger\ddagger}$.

La correction et la complétude de la sémantique opérationnelle par rapport à la sémantique en logique linéaire est établie par le théorème suivant :

Théorème 4 *Pour tout programme* CHRat($\mathcal{X}$) P *et toute requête* $q \in \mathcal{L}'$:

$$\downarrow \mathcal{O}^a_P[q] = \mathcal{LL}_P[q]$$

5 Transformation de programme de CHRat vers CHR

Définition 5 *Soit* $[\![\cdot]\!]$: CHRat($\mathcal{X}$) $\to$ CHR($\mathcal{X}$) *le morphisme suivant, où* $G_u = \{t_1(\overrightarrow{v_1}), \ldots, t_m(\overrightarrow{v_m})\}$, $V_G = \{y_1, \ldots, y_p\}$ *et* $V(H, H') = \{x_1, \ldots, x_n\}$:

$$[\![H \backslash H' \Leftrightarrow G | B_{\mathcal{X}}, B_u]\!] = \begin{cases} H^*, H'^* \Rightarrow G_{\mathcal{X}} | \mathrm{id}\_i(K, x_1, \ldots, x_n), \\ \quad \mathrm{exists}(K, y_1), \ldots, \mathrm{exists}(K, y_p), \\ \quad \mathrm{ask}\_t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{ask}\_t_m(K, \overrightarrow{v_m}). \\ H^* \backslash H'^*, \mathrm{id}\_i(K, x_1, \ldots, x_n), \\ \quad \mathrm{entailed}\_t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{entailed}\_t_m(K, \overrightarrow{v_m}) \\ \quad \Rightarrow G_{\mathcal{X}} | B_{\mathcal{X}}, B^*_u. \end{cases}$$

Les contraintes utilisateur ask$_t_j$, exists, *tokentailed*$\_t_j$ et id$_i$ correspondent aux propositions atomiques introduites dans la sémantique déclarative. i est un identificateur unique associé à la règle. La sémantique en logique linéaire établit le lien entre la sémantique opérationnelle et la transformation définie ci-dessus. Cela conduit au résultat suivant qui prouve la correction de la transformation en sémantique logique linéaire :

Théorème 5 *Pour tout programme* CHRat($\mathcal{X}$) P *et requête* $T_0 \wedge c_0 \in \mathcal{L}'$:

$$\mathcal{LL}_P[T_0 \wedge c_0] = \mathcal{LL}_{[\![P]\!]}(T_0, C_0)$$

6 Une définition hiérarchique d'un solveur de contraintes sur les termes rationnels

6.1 Composant pour la contrainte union-find

L'algorithme classique du *union-find* (ou d'union d'ensembles disjoints) [19] a été implémenté en CHR [18] avec sa meilleure complexité algorithmique connue (temps quasi-linéaire). Atteindre cette complexité est un résultat notable pour un langage déclaratif, en particulier dans le domaine de la programmation logique [11]. L'algorithme *union-find* maintient une partition d'un univers, de sorte que chaque classe d'équivalence a un élément représentatif. Trois opérations agissent sur cette structure de données :

- `make(X)` ajoute l'élément `X` à l'univers, initialement au sein d'une classe réduite au singleton {`X`}.
- `find(X)` renvoie le représentant de la classe d'équivalence de `X`.
- `union(X,Y)` joint la classe d'équivalence de `X` avec celle de `Y` (en changeant éventuellement de représentant).

En tant que solveur de contraintes en logique classique, un tel algorithme résout la contrainte `A` $\simeq$ `B`. Les contraintes CHR `union` et `find` reflètent l'interprétation impérative de la structure de donnée du *union-find*, qui explicite les éléments représentatifs. Cependant, la propriété d'être un élément représentatif est une propriété non monotone qui ne peut pas être capturée par des contraintes en logique classique.

Implantation naïve en CHRat

Une première implantation repose sur une représentation classique des classes d'équivalence par des arbres enracinés [18]. Les racines sont les éléments représentatifs, elles sont marquées comme telles par la contrainte CHR `root(X)`. Les branches de l'arbre sont marquées par `A` $\rightsquigarrow$ `B`, où `A` est l'enfant et `B` le nœud parent.

```
component naive_union_find_solver.
export make/1, ≃/2.
make(A) <=> root(A).
union(A, B) <=>
    find(A, X), find(B, Y), link(X, Y).
A ⇝ B \ find(A, X) <=> find(B, X).
root(A) \ find(A, X) <=> X = A.
link(A, A) <=> true.
link(A, B), root(A), root(B) <=>
    B ⇝ A, root(A).
```

Cette implantation suppose que les points d'entrée `make` and `union` soit utilisés avec des arguments constants seulement, et que le premier argument de `find` soit toujours constant.

Poser la contrainte `A ≃ B` dans le store conduit à l'union des deux classes d'équivalence :

```
A ≃ B ==> union(A, B).
```

Une solution naïve d'implanter le solveur d'implication de ≃ consiste à suivre les branches jusqu'à trouver éventuellement un ancêtre commun pour `A` et `B`.

```
ask(K, A ≃ A) <=> entailed(K, A ≃ A).
A ⇝ C \ ask(K, A ≃ B) <=> C ≃
B | entailed(K, A ≃ B).
B ⇝ C \ ask(K, A ≃ B) <=> A ≃
C | entailed(K, A ≃ B).
```

Le calcul requis pour vérifier l'implication de contraintes est effectué en utilisant les gardes `C ≃ B` et `A ≃ C` récursivement. Soit S l'ensemble des éléments clos de $\mathcal{X}$.

Définition 6 *Un store c* est une structure valide pour *union-find si* `≃/2` *et* `root/1` *forme une forêt d'éléments dans* S.

Proposition 1 *La propriété de validité de la structure d'*union-find *est préservée par toutes les règles du solveur.*

Proposition 2 *Pour tout store* CHR *c avec une structure d'*union-find *valide et pour tous les éléments* `A` *et* `B`*,* `A` *et* `B` *partagent la même classe d'équivalence si et seulement si, pour toute variable fraîche* `K`*,* `entailed(K, A ≃B)` $\in \mathcal{O}_a($`ask(K, A ≃B)` $\wedge c)$.

Implantation optimisée en CHRat

La seconde implantation proposée dans [18] effectue les deux optimisations de *compression de chemins* (*path-compression*) et d'*union par rangs* (*union-by-rank*) pour atteindre la complexité quasi-linéaire en $O(n\alpha(n))$.

```
component union_find.
export make/1, ≃/2.
make(A) <=> root(A, 0).
union(A, B) <=>
    find(A, X), find(B, Y), link(X, Y).
A ⇝ B, find(A, X) <=> find(B, X), A ⇝
X.
root(A, _) \ find(A, X) <=> X = A.
link(A, A) <=> true.
link(A, B),root(A, N),root(B, M)<=>N≥M|
    B ⇝ A, N1 is max(M+1, N), root(A, N1).
link(B, A),root(A, N),root(B, M)<=>N≥M|
    B ⇝ A, N1 is max(M+1, N), root(A, N1).
```

L'implication optimisée pour la vérification de l'implication repose sur `find` pour trouver efficacement les représentants et ensuite les comparer. `check(K, A, B, X, Y)` représente la connaissance du fait que les représentants des classes d'équivalence de `A` et `B` sont les racines `X` et `Y` respectivement. Quand `X` et `Y` sont connus comme égaux, `entailed(K)` est posé dans le store :

```
ask(K, A ≃ B) <=>
    find(A, X), find(B, Y),
    check(K, A, B, X, Y).
root(X) \ check(K, A, B, X, X) <=>
    entailed(K).
```

Ces deux règles ne suffisent pas à définir un solveur complet pour l'implication parce que la structure d'arbres peut changer. En particulier, les racines trouvées pour `A` et `B` peuvent être invalidées par des appels ultérieurs à `union`, qui transformeront ces racines en sous-nœuds. Quand une ancienne racine devient un sous-nœud, les deux règles suivantes posent à nouveau `find` pour obtenir la nouvelle racine :

```
X ⇝ C \ check(K, A, B, X, Y) <=>
        find(A, Z), check(K, A, B, Z, Y).
Y ⇝ C \ check(K, A, B, X, Y) <=>
        find(B, Z), check(K, A, B, X, Z).
```

6.2 Composant pour la contrainte d'égalité entre arbres rationnels

Considérons maintenant les arbres rationnels, c'est-à-dire les arbres enracinés, ordonnés, non bornés, étiquetés et éventuellement infinis, avec un nombre fini de sous-arbres structurellement distincts [6]. Les nœuds sont supposés appartenir à l'univers considéré par le solveur *union-find*. Deux nœuds appartenant à la même classe d'équivalence sont supposés être structurellement égaux. Chaque nœud `X` a une signature `F/N` où `F` est l'étiquette de `X` et `N` est son arité : la contrainte associée est notée `fun(X, F, N)`. Pour chaque `I` entre 1 et `N`, la contrainte `arg(X, I, Y)` énonce que le `I`ème sous-arbre de `X` est (structurellement égal à) `Y`. Ces contraintes ont seulement à être compatible entre les éléments d'une même classe d'équivalence, ce que forcent les règles suivantes :

```
component rational_tree_solver.
import ≃/2 from union_find_solver.
export fun/3, arg/3, ∼/2.
fun(X0, F0, N0) \ fun(X1, F1, N1) <=>
    X0 ≃ X1 |
    F0 = F1, N0 = N1.
arg(X0, N, Y0) \ arg(X1, N, Y1) <=>
    X0 ≃ X1 |
    Y0 ≃ Y1.
```

La contrainte que deux arbres sont structurellement égaux, notée `X ∼ Y`, est réduite à l'union des deux classes d'équivalence :

```
X ∼ Y <=> X ≃ Y.
```

Le calcul associé à la vérification de l'implication de `A` $\sim$ `B` requiert une dérivation co-inductive des comparaisons structurelles pour casser les boucles. Ceci est fait par *mémoisation* : les jetons `checking(K, A, B)` signalent que `A` peut être supposé égal à `B` lorsque la vérification de cette égalité est déjà en cours. `checkTreeAux` vérifie que les signatures de `A` et de `B` sont égales et compare les arguments.

```
ask(K, A ∼ B) <=> checkTree(K, A, B).
checkTree(K, A, B) <=> eqTree(K, A, B) |
    entailed(K, A ∼ B).
ask(eqTree(K, A, B)) <=>
    checking(K, A, B),
    fun(A, FA, NA), fun(B, FB, NB),
    checkTreeAux(K, A, B, FA, NA, FB, NB).
checkTreeAux(K, A, B, F, N, F, N) <=>
    askArgs(K, A, B, 1, N),
    collectArgs(K, A, B, 1, N).
```

`askArgs` ajoute un jeton `askArg` par pair point-à-point de sous-arbres de `A` et de `B`. `askArg` répond `entailedArg` si tous correspondent. `collectArg` assure que tous les jetons `entailedArg` ont été posés avant de conclure sur l'implication de `eqTree(K, A, B)`.

```
askArgs(K, A, B, I, N) <=> I ≤ N |
    arg(A, I, AI), arg(B, I, BI),
    askArg(K, A, B, I, AI, BI),
    J is I + 1, askArgs(K, A, B, J, N).
askArgs(K, A, B, I, N) <=> true.
collectArgs(K, A, B, I, N),
    entailedArg(K, A, B, I) <=>
    J is I + 1,
    collectArgs(K, A, B, J, N).
collectArgs(K, A, B, I, N) <=> I > N |
    entailed(eqTree(K, A, B)).
```

`askArg` vérifie en premier lieu si l'égalité a été *mémoisée*, et sinon demande sa vérification :

```
checking(K, AI, BI) \
   askArg(K, A, B, I, AI, BI) <=>
   entailedArg(K, A, B, I).
askArg(K, A, B, I, AI, BI) <=>
   eqTree(K, AI, BI) |
   entailedArg(K, A, B, I).
```

Par simplicité, ce programme n'a pas de ramasse-miette. En particulier, les jetons mémoisés `checking(K,A,B)` ne sont jamais retirés du store, et les contraintes non impliquées ne sont pas éliminées.

7 Conclusion et perspectives

Nous avons montré qu'en laissant le programmeur définir en CHRat non seulement la vérification de satisfiabilité mais aussi la vérification d'implication de contraintes, cette version de CHR devient complètement modulaire, c'est-à-dire que les composants définis peuvent être réutilisés dans les règles et les gardes d'autres composants sans restriction. De plus, cette discipline de programmation n'est pas trop coûteuse pour le programmeur, le solveur peut se réduire à une simple introspection du store par `c \ ask(K, c) <=> entailed(K)`. Dans le cas général cependant, les règles CHRat pour `ask(K,c)` peuvent effectuer des calculs complexes arbitraires pour dériver le jeton de contraintes `entailed(K)`.

Les sémantiques opérationnelles et déclaratives de CHRat en logique linéaire ont été définies et prouvées équivalentes, et la transformation de programme de CHRat en CHR qui est à la base de notre compilateur a été prouvée correcte vis-à-vis de la sémantique en logique linéaire. Il est bon de noter que la transformation décrite est dirigée par la syntaxe (uniforme) et donc compatible avec d'autres préoccupations orthogonales concernant la modularité, comme les méthodologies pour faire collaborer plusieurs solveurs de contraintes [2]. Nous avons aussi montré que certains exemples classiques de solveurs de contraintes définis en CHR peuvent facilement être modularisés en CHRat et réutilisés pour construire des solveurs de contraintes complexes.

Comme perspectives futures, le cadre fourni par CHRat peut être amélioré de plusieurs façons. Les solveurs d'implication ne devraient jamais conduire à un échec et ne devraient pas interférer avec l'interprétation logique des contraintes exportées présentes dans le store CHR. L'exemple du *union-find* est un cas typique où le solveur d'implication change le store par compression de chemins tout en conservant la contrainte exportée $\simeq$ inchangée. La transformation d'un programme CHRat en un programme CHR classique fait que notre implantation bénéficie directement des optimisations des compilateurs CHR et peut suggérer de nouvelles optimisations. Alors que la gestion efficace des `ask` et `entailed` est laissée à l'implantation CHR sous-jacente, la gestion de la mémoire, du cache et de la mémoisation pour la vérification de l'implication est laissée au programmeur. De bonnes stratégies pour la récupération des miettes ainsi que pour la vérification de la non-implication restent à trouver, comme le montre le solveur pour les arbres rationnels. Enfin, le problème de compilation séparée n'a pas été discuté ici mais constitue un prolongement naturel.

Remerciements

Nous remercions Rishi Kumar pour son travail préliminaire dans cette voie avec le premier auteur, et Jacques Robin pour son récent intérêt pour ce travail.

Références

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266, Linz, 1997. Springer-Verlag.

[2] Slim Abdennadher and Thom W. Frühwirth. Integration and optimization of rule-based constraint solvers. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation. LOPSTR, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2003.

[3] Slim Abdennadher and Heribert Schütz. CHRv : A flexible query language. In *FQAS '98 : Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.

[4] Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In *Proceeding of CP 2005, 11th*, pages 137–151, 2005.

[5] Emmanuel Coquery and François Fages. A type system for CHR. In *Recent Advances in Constaints, revised selected papers from CSCLP'05*, number 3978 in Lecture Notes in Artificial Intelligence, pages 100–117. Springer-Verlag, 2006.

[6] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25 :95–169, 1983.

[7] Gregory Duck, Maria Garcia de la Banda, and Peter Stuckey. Compiling ask constraints. In *Proceedings of International Conference on Logic Programming ICLP 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[8] Gregory J. Duck, Peter J. Stuckey, Marìa Garcìa de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of PPDP'03, International Conference on Principles and Practice of Declarative Programming, Uppsala, Sweden*, pages 79–90. ACM Press, 2003.

[9] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming : operational and phase semantics. *Information and Computation*, 165(1) :14–41, February 2001.

[10] T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3) :95–138, October 1998.

[11] Harald Ganzinger. A new metacomplexity theorem for bottom-up logic programs. In *Proceedings of International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2001.

[12] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[13] K.P. Girish and Sunil Jacob John. Relations and functions in multiset context. *Information Sciences*, 179(6) :758–768, 2009.

[14] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[15] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3) :139–164, 1998.

[16] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[17] T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Fruhwirth. Automatic implication checking for CHR constraint solvers. *Electronic Notes in Theoretical Computer Science*, 147 :93–111, January 2006.

[18] Tom Schrijvers and Thom W. Frühwirth. Analysing the CHR implementation of unionfind. In *Proceedings of the 19th Workshop on (Constraint) Logic Programming, WCLP'05*, Ulm, Germany, 2005.

[19] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of ACM*, 31(2) :245–281, April 1984.